Improved static symmetry breaking for SAT

Jo Devriendt[1], Bart Bogaerts[2,1], Maurice Bruynooghe[1], Marc Denecker[1]

[1] KU Leuven – University of Leuven
Celestijnenlaan 300A, Leuven, Belgium
firstname.lastname@cs.kuleuven.be
[2] Helsinki Institute for Information Technology HIIT
Department of Computer Science, Aalto University, FI-00076 AALTO, Finland

Abstract. An effective SAT preprocessing technique is the construction of symmetry breaking formulas: auxiliary clauses that guide a SAT solver away from needless exploration of symmetric subproblems. However, during the past decade, state-of-the-art SAT solvers rarely incorporated symmetry breaking. This suggests that the reduction of the search space does not outweigh the overhead incurred by detecting symmetry and constructing symmetry breaking formulas. We present three methods to construct more effective symmetry breaking formulas. The first method simply improves the encoding of symmetry breaking formulas. The second detects special symmetry subgroups, for which complete symmetry breaking formulas exist. The third infers binary symmetry breaking clauses for a symmetry group as a whole rather than longer clauses for individual symmetries. We implement these methods as a symmetry breaking preprocessor, and verify their effectiveness on both hand-picked problems as well as the 2014 SAT competition benchmark set. Our experiments indicate that our symmetry breaking preprocessor improves the current state-of-the-art in static symmetry breaking for SAT and has a sufficiently low overhead to improve the performance of modern SAT solvers on hard combinatorial instances.

1 Introduction

Hard combinatorial problems often exhibit symmetry. Eliminating symmetry potentially boosts solver performance as it prevents a solver from needlessly exploring isomorphic parts of a search space. One common method to eliminate symmetries is to add symmetry breaking formulas to the problem specification [6, 1], which is called *static symmetry breaking*. For the Boolean satisfiability problem (SAT), the state-of-the-art tool SHATTER [3] implements this technique; it functions as a preprocessor that can be used with any SAT solver. *Dynamic symmetry breaking*, on the other hand, interferes in the search process by adding symmetric versions of learned clauses [23, 8] or by avoiding symmetric choices [21].

Symmetry properties of a SAT problem are typically detected by first converting the problem to a colored graph such that the graph's automorphism group corresponds to a symmetry group of the SAT problem and subsequently calling a graph automorphism tool such as NAUTY [19], SAUCY [15] or BLISS [14] to find automorphism groups.

Static symmetry breaking proceeds by adding formulas that exclude symmetric assignments. A symmetry breaking formula is *sound* if it preserves at least one assignment

from each class of symmetric assignments and *complete* if it preserves at most one assignment from each class. One sound symmetry breaking constraint is the *lex-leader* constraint, which holds exactly for those assignments lexicographically smaller than their symmetric image [6]. The conjunction of lex-leader constraints for every symmetry in a symmetry group constitutes a complete symmetry breaking constraint for that group. However, symmetry groups tend to be too large to enforce lex-leader for each symmetry. Instead, *partial* symmetry breaking adds lex-leader constraints only for a set of generators of the group [18]. While effective for many symmetric problems, the pruning power of partial symmetry breaking depends heavily on the chosen set of generators and whether or not compositions of these generators are eliminated as well [16].

In this paper, we present BREAKID,[3] a symmetry breaking preprocessor for SAT that follows in SHATTER's footsteps. BREAKID sports several improvements compared to SHATTER, ranging from small "hacks" to avoid known problems, as well as novel ideas to exploit a symmetry group's structure. Three of these improvements are investigated in-depth in this paper. Firstly, we evaluate a *compact CNF encoding* of the lex-leader constraint. Secondly, we show how to detect *row interchangeability symmetry* subgroups, for which a small set of generators exists such that their lex-leader constraints do break the subgroup completely. Thirdly, we show how to generate *binary symmetry breaking clauses* not based on individual generators, but on algebraic properties of the entire symmetry group. A common theme in the second and third improvement is to simultaneously adjust the set of generators and the variable ordering, both needed to construct lex-leader constraints.

We evaluate the proposed symmetry breaking improvements individually, and verify the effectiveness of both SHATTER and BREAKID on 2014's SAT competition instances. From our experiments, we conclude that (i) more compact CNF encodings have a small impact on runtime and memory consumption, (ii) row-interchangeability detection improves performance on several problems, (iii) group-based binary symmetry breaking clauses are effective for a particular class of problems, (iv) BREAKID outperforms SHATTER on most benchmarks, and (v) symmetry breaking leads to significant performance gains on the hard combinatorial instances of 2014's SAT competition.

After some preliminaries in Section 2, we present our improvements in Sections 3, 4 and 5 respectively. Section 6 describes our symmetry breaking preprocessor BREAKID and Section 7 contains the experimental evaluation. We conclude in Section 8.

2 Preliminaries

Satisfiability problem Let Σ be a set of Boolean variables and $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$ the set of Boolean values. For each $x \in \Sigma$, there exist two *literals*; the *positive* literal denoted by x and the *negative* literal denoted by $\neg x$. The set of all literals over Σ is denoted $\overline{\Sigma}$. A *clause* is a finite disjunction of literals, and a *formula* is a finite conjunction of clauses (as usual, we assume formulas are in conjunctive normal form (CNF)). An *assignment* α is a mapping $\Sigma \to \mathbb{B}$. We extend α to literals as $\alpha(\neg x) = \neg\alpha(x)$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$. An assignment satisfies a formula iff at least one literal from each clause

[3] Pronounced "Break it!".

is mapped to $\mathbf{t}$ by α. The Boolean Satisfiability (SAT) problem consists of deciding whether there exists an assignment that satisfies a propositional formula.

Group theoretical concepts A *permutation* is a bijection from a set to itself. We write permutations in *cycle notation*: $(abc)(de)$ is the permutation that maps a to b, b to c, c to a, swaps d with e, and maps all other elements to themselves. A *swap* is a non-trivial permutation that is its own inverse. Permutations form algebraic groups under the composition relation ($\circ$). A set of permutations P is a set of *generators* for a permutation group Π if each permutation in Π is a composition of permutations from P. The group $Grp(P)$ is the permutation group *generated* by all compositions of permutations in P. The *support* $supp(\pi)$ of a permutation π is the set of elements $\{x \mid \pi(x) \neq x\}$. The support $supp(\Pi)$ of a permutation group Π is the union of the supports of permutations in Π. The *orbit* $Orb_\Pi(x)$ of an element x under a permutation group Π is the set $\{\pi(x) \mid \pi \in \Pi\}$. A group Π *stabilizes* an element x if $x \notin supp(\Pi)$. The *stabilizer subgroup* $Stab_\Pi(x)$ of a permutation group Π for an element x is the group $\{\pi \in \Pi \mid \pi(x) = x\}$, or equivalently, the largest subgroup of Π that stabilizes x.

Symmetry in SAT Let π be a permutation of $\overline{\Sigma}$. We extend π to clauses: $\pi(l_1 \vee \ldots \vee l_n) = \pi(l_1) \vee \ldots \vee \pi(l_n)$; to formulas: $\pi(c_1 \wedge \ldots \wedge c_n) = \pi(c_1) \wedge \ldots \wedge \pi(c_n)$; to assignments: $\pi(\alpha)(l) = \alpha(\pi(l))$. A *symmetry* of a formula ϕ is a permutation π of $\overline{\Sigma}$ that *commutes with negation* (i.e., $\pi(\neg l) = \neg\pi(l)$) and that *preserves satisfaction* to ϕ (i.e., $\pi(\alpha)$ satisfies ϕ iff α satisfies ϕ). A permutation of literals π that commutes with negation and for which $\pi(\phi) = \phi$ is a *syntactical* symmetry of ϕ. Typically, only syntactical symmetry is exploited, since this type of symmetry can be detected with relative ease. The practical techniques presented in this paper are no exception, though all presented theorems hold for non-syntactical symmetry as well.

Symmetry breaking Symmetry *breaking* aims at eliminating symmetry, either by *statically* posting symmetry breaking constraints that invalidate symmetric assignments, or by altering the search space *dynamically* to avoid symmetric search paths. A (static) symmetry breaking formula for SAT is presented in Section 3. If Π is a symmetry group, then a symmetry breaking formula ψ is *sound* if for each assignment α there exists at least one symmetry $\pi \in \Pi$ such that $\pi(\alpha)$ satisfies ψ; ψ is *complete* if for each assignment α there exists at most one symmetry $\pi \in \Pi$ such that $\pi(\alpha)$ satisfies ψ [27].

3 Compact CNF encodings of the lex-leader constraint

A classic approach to static symmetry breaking is to construct *lex-leader constraints*.

Definition 1 (Lex-leader constraint [6]). *Let ϕ be a formula over Σ, π a symmetry of ϕ, $\preceq_x$ an order on Σ and $\preceq_\alpha$ the induced lexicographic order on the set of assignments over Σ. A formula LL_π over $\Sigma' \supseteq \Sigma$ is a* lex-leader constraint *for π if for each Σ-assignment α, there exists a Σ'-extension of α that satisfies LL_π iff $\alpha \preceq_\alpha \pi(\alpha)$.*

In other words, each assignment whose symmetric image under π is smaller, is eliminated by LL_π. It is easy to see that the conjunction of LL_π for all π in some $\Pi' \subseteq \Pi$ is a sound (but not necessarily complete) symmetry breaking constraint for Π.

An efficient encoding of the lex-leader constraint LL_π as a conjunction of clauses is given by Aloul et al. [1], where each variable in $supp(\pi)$ leads to 2 clauses of size 3 and 2 clauses of size 4. Below, we give a derivation of a more compact encoding of LL_π as a conjunction of 3 clauses of size 3 for each variable in $supp(\pi)$, which is more compact and hence reduces the overhead introduced by posting it. A similar encoding is presented by Sakallah [22] but has not been experimentally evaluated before.

Theorem 1 (Compact encoding of lex-leader constraint). *Let π be a symmetry, let $supp(\pi) = \{x_1, \ldots, x_n\}$ be ordered such that $x_i \preceq_x x_j$ iff $i \leq j$ and let $\{y_0, \ldots, y_{n-1}\}$ be a set of auxiliary variables disjoint from $supp(\pi)$. The following set of clauses is a lex-leader constraint for π:*

$$\begin{array}{rl|rl} y_0 & & y_j \vee \neg y_{j-1} \vee \neg x_j & 1 \leq j < n \\ \neg y_{i-1} \vee \neg x_i \vee \pi(x_i) & 1 \leq i \leq n & y_j \vee \neg y_{j-1} \vee \pi(x_j) & 1 \leq j < n \end{array}$$

Proof. Crawford et al. proposed the following lex-leader constraint [6]:

$$\forall i : (\forall j < i : x_j \Leftrightarrow \pi(x_j)) \Rightarrow \neg x_i \vee \pi(x_i) \tag{1}$$

Assuming $\mathbf{f} < \mathbf{t}$, this constraint expresses that the value of a variable x_i must be less than or equal to the value of $\pi(x_i)$ if for all smaller variables x_j, x_j has the same value as $\pi(x_j)$. As such, it encodes a valid lex-leader constraint.

Aloul et al. [1] noticed that the antecedent $(\forall j < i : x_j \Leftrightarrow \pi(x_j))$ is recursively reified by introducing auxiliary variables y_j:

$$y_j \Leftrightarrow (y_{j-1} \wedge (x_j \Leftrightarrow \pi(x_j))) \tag{2}$$

where the base case y_0 is fixed to be true. In essence, y_j holds iff x_k and $\pi(x_k)$ have the same truth value for $1 \leq k \leq j$. Equation (1) then translates to:

$$y_0 \tag{3}$$
$$y_j \Leftrightarrow (y_{j-1} \wedge (x_j \Leftrightarrow \pi(x_j))) \qquad 1 \leq j < n \tag{4}$$
$$y_{i-1} \Rightarrow \neg x_i \vee \pi(x_i) \qquad 1 \leq i \leq n \tag{5}$$

Note that by (5), if y_{j-1} holds, then $\neg x_j \vee \pi(x_j)$ holds. Hence, $y_{j-1} \wedge (x_j \Leftrightarrow \pi(x_j))$ simplifies to $y_{j-1} \wedge (x_j \vee \neg\pi(x_j))$, and (4) simplifies to:

$$y_j \Leftrightarrow (y_{j-1} \wedge (x_j \vee \neg\pi(x_j))) \qquad 1 \leq j < n \tag{4'}$$

Lastly, we observe that y_j only occurs negatively in formula (5). Thus, only one implication in the definition of y_j is important for the correctness of this constraint. Relaxing the constraints when y_j must be false leads to:

$$y_j \Leftarrow (y_{j-1} \wedge (x_j \vee \neg\pi(x_j))) \qquad 1 \leq j < n \tag{4''}$$

Working out the implications and applying distributivity of $\wedge$ and $\vee$ in equations $(3, 4'', 5)$ leads to the CNF formula in this theorem and hence, concludes our proof. □

The relaxation introduced by step 4" does not weaken the symmetry breaking capacity of our encoding, as it only weakens the constraints on auxiliary variables not permuted by any original symmetry. However, 4" still represents weaker constraints than 4'. In Section 7 we give experimental results that compare the presented compact encoding with an "unrelaxed" clausal encoding based on 4', having an extra binary and ternary clause. It turns out that the relaxation of 4" leads to slightly less overhead.

Note that the condition y_j is satisfied in fewer assignments as j increases, so the marginal effect of posting the above constraints decreases as j increases. Still, the marginal cost is stable at three clauses and one auxiliary variable regardless of j. Because of this, the size of lex-leader constraints is often limited by putting an upper bound k on the number of auxiliary variables to be introduced [2], resulting in a shorter lex-leader constraint LL^k_π. BREAKID also employs this limit on the size of its lex-leader constraints, by default posting a conservative LL^{50}_π for each generator symmetry π.

4 Exploiting row interchangeability

An important type of symmetry is *row interchangeability*, which is present when a subset of variables can be structured as a two-dimensional matrix such that each permutation of the rows induces a symmetry. This form of symmetry is common; often it stems from an interchangeable set of objects in the original problem domain, with each row of variables expressing certain properties of one particular object. Examples are intercheangeability of pigeons or holes in a pigeonhole problem, interchangeability of nurses in a nurse scheduling problem, fleets of interchangeable trucks in a delivery system etc. Exploiting this type of matrix symmetry with adjusted symmetry breaking techniques can significantly improve SAT performance [8, 17]. In this section, we present a novel way of automatically dealing with row interchangeability in SAT.

Example 1 (Row interchangeability in graph coloring). Let ϕ be a CNF formula expressing the graph coloring constraint that two directly connected vertices cannot have the same color. Let $\Sigma = \{x_{11}, \ldots, x_{nm}\}$ be the set of variables, with intended interpretation that x_{ij} holds iff vertex j has color i. Given the nature of the graph coloring problem, all colors are interchangeable, so each color permutation ρ induces a symmetry π_ρ of ϕ. More formally, the color interchangeability symmetry group consists of all symmetries

$$\pi_\rho : \overline{\Sigma} \to \overline{\Sigma} : x_{ij} \mapsto x_{\rho(i)j}, \neg x_{ij} \mapsto \neg x_{\rho(i)j}$$

If we structure Σ as a matrix where x_{ij} occurs on row i and column j, then each permutation of rows corresponds to a permutation of colors, and hence a symmetry.

Definition 2 (Row interchangeability in SAT [8]). *A* variable matrix M *is a bijection* $M : Ro \times Co \to \Sigma'$ *from two sets of indices* Ro *and* Co *to a set of variables* $\Sigma' \subseteq \Sigma$. *We refer to* $M(r, c)$ *as* x_{rc}. *The* r*'th* row *of* M *is the sequence of variables* $[x_{r1}, \ldots, x_{rm}]$, *the* c*'th* column *is the sequence* $[x_{1c}, \ldots, x_{nc}]$. *A formula* ϕ *exhibits* row interchangeability *symmetry if there exists a variable matrix* M *such that for each permutation* $\rho : Ro \to Ro$

$$\pi^M_\rho : \overline{\Sigma}' \to \overline{\Sigma}' : x_{rc} \mapsto x_{\rho(r)c}, \neg x_{rc} \mapsto \neg x_{\rho(r)c}$$

is a symmetry of ϕ. The row interchangeability symmetry group of a matrix M is denoted as R_M.

A useful property of row interchangeability is that it is broken completely by only a linearly sized symmetry breaking formula [10, 25].

Theorem 2 (Complete symmetry breaking for row interchangeability [8]). *Let ϕ be a formula and R_M a row interchangeability symmetry group of ϕ with $Ro = \{1, \ldots, n\}$ and $Co = \{1, \ldots, m\}$. If the total variable order $\preceq_{\mathrm{x}}$ on Σ satisfies $x_{ij} \preceq_{\mathrm{x}} x_{i'j'}$ iff $i < i'$ or ($i = i'$ and $j \leq j'$), then the conjunction of lex-leader constraints for $\pi^M_{(k\ k+1)}$ with $1 \leq k < n$ breaks M_R completely.*

In words, Theorem 2 guarantees that for a natural ordering of the variable matrix, the lex-leader constraints for the swap of each two subsequent rows form a complete symmetry breaking formula for row interchangeability. The condition that the order "matches" the variable matrix is important: the theorem no longer holds without it.

If we are able to detect that a formula exhibits row interchangeability, we can break it completely by choosing the right order and posting the right lex-leader constraints. In practice, symmetry detection tools for SAT only present us with a set of generators for the symmetry group, which contains no information on the *structure* of this group. The challenge at hand is to derive row interchangeability subgroups from these generators.

4.1 Row interchangeability detection algorithm

Given a set of generators P for a symmetry group Π of a formula ϕ, the task is to detect a variable matrix M that represents a row interchangeability subgroup $R_M \subseteq \Pi$. We present an algorithm that is sound, but incomplete in the sense that it does not guarantee that all row interchangeability subgroups present are detected.

The first step is to find an initial row interchangeable variable matrix M consisting of three rows. This is done by selecting two swap symmetries π_1 and π_2 that represent two swaps of three rows. More formally, suppose π_1 and π_2 are such that (with $r = supp(\pi_1) \cap supp(\pi_2)$) the following three conditions hold (i) $\pi_1 = \pi_1^{-1}$ and $\pi_2 = \pi_2^{-1}$, (ii) r, $\pi_1(r)$ and $\pi_2(r)$ are disjoint, and (iii) $supp(\pi_1) = r \cup \pi_1(r)$ and $supp(\pi_2) = r \cup \pi_2(r)$. In this case, r, $\pi_1(r)$ and $\pi_2(r)$ form three rows of a row interchangeable variable matrix, and π_1 and π_2 are swaps of those rows.

If, after inspecting all pairs of swaps in P, no three-rowed matrix is found, the algorithm stops, in which case we do not know whether a row interchangeability subgroup exists. However, our experiments indicate that for many problems, an initial three-rowed matrix can be derived from a detected set of generator symmetries.

The second step maximally extends the initial variable matrix M with new rows. The idea is that for each symmetry $\pi \in P$ and each row r of M, $\pi(r)$ is a candidate row to add to M. This is the case if $\pi(r)$'s literals are disjoint from M's literals and swapping $\pi(r)$ with r is a syntactical symmetry of ϕ.

Pseudocode is given in Algorithm 1. This algorithm terminates since both P and the number of rows in any row interchangeability matrix are finite. The algorithm is sound: each time a row is added, it is interchangeable with at least one previously added row and hence, by induction, with all rows in M. If k is the largest support size of a

input : P, ϕ
output: M
identify two swaps $\pi_1, \pi_2 \in P$ that induce an initial variable matrix M with 3 rows;
repeat
  foreach *permutation* π *in* P **do**
    foreach *row* r *in* M **do**
      if $\pi(r)$ *is disjoint from* M *and swapping* r *and* $\pi(r)$ *is a symmetry of* ϕ **then**
        add $\pi(r)$ as a new row to M;
      end
    end
  end
until *no extra rows are added to* M;
return M;

Algorithm 1: Row interchangeability detection

symmetry in P, then finding an initial row interchangeable matrix based on two row swap symmetries in P takes $O(|P|^2 k)$ time. With an optimized implementation that avoids duplicate combinations of generators and rows, extending the initial matrix with extra interchangeable rows has a complexity of $O(|P||Ro||\phi|k)$, with Ro the set of row indices of M. Algorithm 1 then has a complexity of $O(|P|^2 k + |P||Ro||\phi|k)$.

As mentioned before, the algorithm is not complete: it might not be possible to construct an initial matrix, or even given an initial matrix, there is no guarantee to detect all possible row extensions, as only the set of generators instead of the whole symmetry group is used to calculate a new candidate row.

It is straightforward to extend Algorithm 1 to detect multiple row interchangeability subgroups. After detecting a first row interchangeability subgroup R_M, remove any generators from P that also belong to R_M. This can be done by standard algebraic group membership tests, which are efficient for interchangeability groups [24]. Then, repeat Algorithm 1 with the reduced set of generator symmetries until no more row interchangeability subgroups are detected.

Example 2 (Example 1 continued.). Let φ and the x_{ij} be as in Example 1. Suppose we have five colors and three vertices. Vertex 1 is connected to vertex 2 and 3; vertices 2 and 3 are not connected. This problem has a symmetry group Π induced by the interchangeability of the colors and by a swap on vertex 2 and 3. A set of generators for Π is $\{\pi_{(12)}, \pi_{(23)}, \pi_{(34)}, \pi_{(45)}, \nu_{23}\}$, where $\pi_{(ij)}$ is the symmetry that swaps colors i and j (as in the previous example), and ν_{23} is the symmetry obtained by swapping vertices 2 and 3, i.e.,[4]

$$\nu_{23} = (x_{12}\ x_{13})(x_{22}\ x_{23})(x_{32}\ x_{33})(x_{42}\ x_{43})(x_{52}\ x_{53}).$$

For these generators, it is obvious that the $\pi_{(ij)}$ form a row interchangeability matrix. However, a symmetry detection tool might return the alternative set of symmetry

[4] We omit negative literals from the cycle notation; a symmetry always commutes with negation.

generators $P = \{\pi_{(12)}, \pi_{(23)}, \sigma_1, \sigma_2, \nu_{23}\}$ with

$$\sigma_1 = \pi_{(35)} \circ \pi_{(13)} = (x_{11}\ x_{31}\ x_{51})(x_{12}\ x_{32}\ x_{52})(x_{13}\ x_{33}\ x_{53})$$
$$\sigma_2 = \pi_{(34)} \circ \nu_{23} = (x_{12}\ x_{13})(x_{22}\ x_{23})(x_{31}\ x_{41})(x_{32}\ x_{43})(x_{42}\ x_{33})(x_{52}\ x_{53}).$$

The challenge is to detect the color interchangeability subgroup starting from P.

The first step of Algorithm 1 searches for two swaps in P that combine to a 3-rowed variable matrix. $\pi_{(12)}$ and $\pi_{(23)}$ fit the bill, resulting in a variable matrix M with rows:

$$[x_{11}, x_{12}, x_{13}] \quad [x_{21}, x_{22}, x_{23}] \quad [x_{31}, x_{32}, x_{33}]$$

Applying σ_1 on the third row results in:

$$[x_{51}, x_{52}, x_{53}]$$

which after a syntactical check on ϕ is confirmed to be a new row to add to M.

Unfortunately, the missing row $[x_{41}, x_{42}, x_{42}]$ is not derivable by applying any generator in P on rows in M, so the algorithm terminates.

The failure of detecting the missing row in Example 2 stems from the fact that the generators σ_1 and σ_2 are obtained by complex combinations of symmetries in the interchangeability subgroup and the symmetry ν_{23}. This inspires a small extension of Algorithm 1. As soon as the algorithm reaches a fixpoint, we call the original symmetry detection tool to search for a set of generators of the subgroup that stabilizes all but one rows of the matrix M found so far. This results in "simpler" generators that do not permute the literals of the excluded rows. Tests on CNF instances show that this simple extension, although giving no theoretical guarantees, often manages to find new generators that, when applied on the current set of rows, construct new rows. After detecting row-stabilizing symmetries, Algorithm 1 resumes from line 2, aiming to extend the matrix further by applying the extended set of generators. This process ends when even the new generators can no longer derive new rows.

Example 3 (Example 2 continued.). The only symmetry of the problem that stabilizes the variables $\{x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}\}$ is $\pi_{(45)}$, which has the missing row $[x_{41}, x_{42}, x_{42}]$ as image of the fifth row.

The matrix, which now contains all variables, allows one to completely break the color interchangeability. The symmetry between vertices 2 and 3 is not expressed in the matrix, but can still be broken by a lex-leader constraint, as described in Section 6.

5 Generating binary symmetry breaking clauses

Partial symmetry breaking, where lex-leader constraints are posted only for a set of generators of a symmetry group Π, is motivated by the infeasibility of posting lex-leader constraints for all symmetries in Π. An alternative we explore here is to post only a very short lex-leader constraint, namely LL^1_π, but do this for a large number of $\pi \in \Pi$. As already mentioned in Section 3, the first parts of the lex-leader constraint

breaks comparatively more symmetry than later parts, so in that sense, posting LL^1_π is the most cost-effective way of breaking π.

Note that LL^1_π is equivalent to the binary clause $\neg x \vee \pi(x)$ where x is the smallest variable in $supp(\pi)$ according to $\preceq_{\mathrm{x}}$. To construct as many of these binary clauses as possible without enumerating the whole symmetry group Π, we use a greedy approach that starts from the generators of Π and exploits the freedom to reorder the variables of Σ as well as the fact that one can easily compute the orbit of a literal in Π.

Theorem 3 (Binary symmetry breaking clauses). *Let Π be a non-trivial symmetry group of ϕ, $\preceq_{\mathrm{x}}$ an ordering of Σ, and x^* the $\preceq_{\mathrm{x}}$-smallest variable in $supp(\Pi)$. For each $x \in Orb_\Pi(x^*)$, the binary clause $\neg x^* \vee x$ is entailed by LL_π for some $\pi \in \Pi$.*

Proof. If $x = x^*$, the theorem is trivially true. If $x \neq x^*$, there exists a $\pi \in \Pi$ with $\pi(x^*) = x$ since $x \in Orb_\Pi(x^*)$. Since x^* is the smallest variable in $supp(\Pi)$, it is also the smallest in $supp(\pi)$. Theorem 1 shows that y_0 and $\neg y_0 \vee \neg x^* \vee \pi(x^*)$ are two clauses in LL_π. Resolving y_0 leads to $\neg x^* \vee \pi(x^*)$, which is equivalent to $\neg x^* \vee x$.

Theorem 3 allows to construct small lex-leader clauses for Π without enumerating individual members of Π; it suffices to compute the orbit of the smallest variable in $supp(\Pi)$ to derive a set of binary symmetry breaking clauses. Theorem 3 holds for all symmetry groups, so also for any subgroup Π' of Π. In particular, if Π' stabilizes the smallest variable in $supp(\Pi)$, applying Theorem 3 to Π' results in different clauses than applying it to Π, as Π' has a different smallest variable in its support.

Example 4. Let $P = \{(ab)(cdef)\}$ and $\Pi = Grp(P) = \{(ab)(cdef), (ab)(cfed), (ce)(df)\}$.[5] With order $a \preceq_{\mathrm{x}} b \preceq_{\mathrm{x}} c \preceq_{\mathrm{x}} d \preceq_{\mathrm{x}} e \preceq_{\mathrm{x}} f$, a is the $\preceq_{\mathrm{x}}$-smallest variable of $supp(\Pi)$. Theorem 3 guarantees that $\neg a \vee b$ is a consequence of the lex-leader constraints for Π. Let $\Pi' = Stab_\Pi(a) = \{(ce)(df)\}$, then c is the $\preceq_{\mathrm{x}}$-smallest variable of $supp(\Pi')$, hence also $\neg c \vee e$ is entailed by the lex-leader constraints for Π.

If we assume a different order $\preceq'_{\mathrm{x}}$, different binary clauses are obtained. For instance, let c be the $\preceq'_{\mathrm{x}}$-smallest variable of $supp(\Pi)$. Then Theorem 3 allows us to post the clauses $\neg c \vee d, \neg c \vee e$ and $\neg c \vee f$ as symmetry breaking clauses. The stabilizer subgroup $Stab_\Pi(c)$ is empty, so no further binary clauses can be derived.

A *stabilizer chain* is a sequence of stabilizer subgroups starting with the full group Π and ending with the trivial group containing only the identity, where each next subgroup in the chain stabilizes an extra element. Given a variable order $\preceq_{\mathrm{x}}$, applying Theorem 3 to each subgroup in a stabilizer chain stabilizing literals according to $\preceq_{\mathrm{x}}$ for a symmetry group Π, is equivalent to constructing all LL^1_π for $\pi \in \Pi$ under $\preceq_{\mathrm{x}}$ [13]. This stabilizer chain idea was used by Puget to efficiently break all-different constraints in a constraint programming context [20].

However, as shown by Example 4, the variable order influences the number of binary symmetry clauses derivable by a stabilizer chain of Π. We present an algorithm that, given a set of generator symmetries P for symmetry group Π, decides a total order on a subset of variables, and constructs binary symmetry breaking clauses for

[5] We again omit negative literals in cycle notation.

input : P
output: LL^{bin},Ord
initialize $Q = P$ and initialize Ord as an empty list;
while $Q \neq \emptyset$ **do**
  O is a largest orbit of $Grp(Q)$;
  x^* is a variable in O for which $\{\pi \in Q \mid \pi(x^*) \neq x^*)\}$ is minimal;
  add x^* to Ord as last variable;
  foreach $x \in O$ **do**
    add $\neg x^* \vee x$ to LL^{bin};
  end
  $Q = \{\pi \in Q \mid \pi(x^*) = x^*\}$;
end
return LL^{bin}, Ord;

Algorithm 2: Binary symmetry breaking clause generation

those variables based on a simultaneously constructed sequence of subgroups stabilizing those variables. The constructed sequence of subgroups stabilizing the literals is no actual stabilizer chain, as each of the subgroups equals $Grp(P')$ for some subset $P' \subseteq P$. The advantage of this approach is simplicity of the algorithm and low computational complexity, although it would be interesting future work to compute an actual stabilizer chain using for instance the *Schreier-Sims algorithm* [24].

In detail, our algorithm starts with an empty variable order Ord and a copy Q of the given set of generators P. It iteratively chooses a suitable variable x^* as next in the variable order, constructs binary clauses based on $Grp(Q)$, and removes any permutations $\pi \in Q$ for which $x \in supp(\pi)$. As a result, at each iteration, $Grp(Q)$ stabilizes all variables in Ord except the last variable x^*, allowing the construction of binary symmetry breaking clauses $\neg x^* \vee x$ for each $x \in Orb_{Grp(Q)}(x^*)$, as per Theorem 3.

A suitable next variable x^* is one that induces a high number of binary symmetry breaking clauses, but removes few symmetries from Q so that the following iterations of the algorithm still have a reasonably sized symmetry group to work with. One way to satisfy these requirements is to pick x^* such that $Orb_{Grp(Q)}(x^*)$ is maximal, and $\{\pi \in Q \mid \pi(x^*) \neq x^*)\}$ is minimal compared to other literals of x^*'s orbit.

Pseudocode is given in Algorithm 2. This algorithm terminates, as while $Q \neq \emptyset$, x^* belongs to a largest orbit of $Grp(Q)$, so $x^* \neq \pi(x^*)$ for at least one $\pi \in Q$. As a result, Q shrinks in size during each iteration, eventually becoming the empty set. The complexity of Algorithm 2 is dominated by finding the largest orbit of $Grp(Q)$, which is $O(|Q||\Sigma|)$, resulting in a total complexity of $O(|P|^2|\Sigma|)$.

Worst case, $O(|supp(\Pi)|^2)$ binary clauses are constructed by Algorithm 2. In particular, if some subgroup Π' of Π represents an interchangeable set of n variables, $n(n-1)/2$ binary clauses are derived. However, in this case Π' also represents a row interchangeability symmetry group, which is completely broken by techniques from Section 4. Performing row interchangeability detection and breaking before binary clause generation can avoid quadratic sets of binary clauses. Section 6 shows this is indeed the order by which BREAKID performs its symmetry breaking.

6 Putting it all together as BREAKID

This section describes how the improvements presented in the previous section combine with eachother and with standard symmetry breaking techniques in the symmetry breaking preprocessor BREAKID.

BREAKID has been around since 2013, when a preliminary version obtained the gold medal in the hard combinatorial sat+unsat track of 2013's SAT competition [5]. This early version incorporated all of SHATTER's symmetry breaking techniques and used a primitive row interchangeability detection algorithm that enumerated symmetries to detect as many row swap symmetries as possible [8]. We developed BREAKID2 in 2015, using the ideas presented in this paper. BREAKID2 entered the main track of 2015's SAT race in combination with GLUCOSE 4.0, placing 10th, ahead of all other GLUCOSE variants. The experiments in the next section are run with a slightly updated version – BREAKID2.1– which has more usability features and reduced memory overhead. For the remainder of this paper, we use BREAKID to refer to the particular implementation BREAKID2.1. BREAKID's source code is published online [7].

6.1 BREAKID's high level algorithm

Preprocessing a formula ϕ by symmetry breaking in BREAKID starts with removing duplicate clauses and duplicate literals in clauses from ϕ, as SAUCY cannot handle duplicate edges. Then, a call to SAUCY constructs an initial generator set P of the syntactical symmetry group of ϕ.

Thirdly, BREAKID detects row interchangeability subgroups R_M of $Grp(P)$ by Algorithm 1. The program incorporates the variables of the support of all R_M in a global variable order Ord such that the conjunction of $LL_{\pi_\rho^M}$ for all subsequent row swaps π_ρ^M under Ord forms a complete symmetry breaking formula for R_M.[6] After adding the complete symmetry breaking formula of each R_M to an initial set of symmetry breaking clauses ψ, we also remove all symmetries in P that belong to some R_M, since these symmetries are broken completely already.

Next, using the pruned P, binary clauses for $Grp(P)$ are constructed by Algorithm 2, which simultaneously decides a set of variables to be smallest under Ord.[7]

Finally, Ord is supplemented with missing variables until it is total, and limited lex-leader constraints LL_π^{50} are constructed for each π left in P. These lex-leader constraints incorporate two extra refinements also used by SHATTER; one for *phase-shifted* variables and one for the *largest variable in a symmetry cycle* [1].

Algorithm 3 gives pseudocode for BREAKID's high-level routine described above.

[6] In case two detected row interchangeability matrices overlap, it is not always possible to choose the order on the variables so that both are broken completely. In this case, one of the row interchangeability groups will only be broken partially.

[7] A small adaptation to Algorithm 2 ensures BREAKID only selects smallest variables that are not permuted by a previously detected row interchangeability group.

input : ϕ
output: ψ
1 remove duplicate clauses from ϕ and duplicate literals from clauses in ϕ;
2 run **SAUCY** to detect a set of symmetry generators P;
3 initialize ψ as the empty formula and Ord as an empty sequence of ordered variables;
4 detect **row interchangeability** subgroups R_M;
5 **foreach** *row interchangeability* R_M *subgroup of* $Grp(P)$ **do**
6 remove $P \cap R_M$ from P;
7 add complete symmetry breaking clauses for R_M to ψ;
8 add $supp(R_M)$ to the back of Ord accordingly;
9 **end**
10 add **binary clauses** for $Grp(P)$ to ψ, add corresponding variables to the front of Ord;
11 add missing variables to the middle of Ord;
12 **foreach** $\pi \in P$ **do**
13 add LL_π^{50} to ψ, utilizing **SHATTER's optimizations**;
14 **end**
15 **return** ψ;

Algorithm 3: Symmetry breaking by BREAKID

7 Experiments

In this section, we verify the effectiveness of the proposed techniques separately, and investigate the feasibility of using BREAKID in the application and hard-combinatorial track of 2014's SAT competition We use eight benchmark sets:

- **app14**: the application track of 2014's SAT competition (300 instances)
- **app14sym**: subset of app14 for which SAUCY detected symmetry (164 instances)
- **hard14**: the hard-combinatorial track of 2014's SAT competition (300 instances)
- **hard14sym**: subset of hard14 for which SAUCY detected symmetry (159 instances)
- **hole**: 8 unsatisfiable pigeonhole instances
- **urquhart**: 6 unsatisfiable Urquhart instances
- **channel**: 10 unsatisfiable channel routing instances
- **color**: 10 unsatisfiable graph coloring instances

Pigeonhole and Urquhart problems are provably hard for purely resolution-based SAT solvers, in the sense that even for very small instances astronomical running time is needed to decide satisfiability of the problem [12, 26]. The employed channel routing and graph coloring instances are highly symmetric, exhibiting strong row interchangeability. They are taken from SYMCHAFF's benchmark set [21]. The graph coloring instances were also used in 2005 and 2007's SAT competitions.

As SAT-solver, we use GLUCOSE 4.0 [4], which is based on MINISAT [9]. We include the symmetry breaking preprocessor SHATTER [3] bundled with SAUCY 3.0 [15] in our experiments. The resources available to each experiment were 10GB of memory and 3600s on an Intel(R) Xeon(R) E3-1225 cpu. The operating system was Ubuntu 14.04 with Linux kernel 3.13. Unless noted otherwise, all results *include* any preprocessing step, such as deduplicating the input CNF, symmetry detection by SAUCY and symmetry breaking clause generation by SHATTER or BREAKID. Detailed experimental results are available online [7].

7.1 Compact symmetry breaking clauses

We first investigate the influence of the compact lex-leader encoding presented in Section 3. The experiment consists of running BREAKID with the *standard* encoding used in SHATTER (four clauses for each variable in a symmetry's support), with BREAKID's default *compact* encoding (three clauses), and with an *unrelaxed* encoding that does not relax the constraints on the auxiliary variables (five clauses). To focus on the difference between the encodings, in this experiment, BREAKID does not exploit row interchangeability, does not generate binary clauses, and does not limit the size of the lex-leader formulas. The benchmark sets employed are app14sym, hard14sym, hole, urquhart, channel and color. The results are presented in Table 1.

Table 1. Number of solved instances for standard, unrelaxed and compact lex-leader encoding, as well as average runtime and memory consumption of GLUCOSE (excluding BREAKID's preprocessing) for solved instances.

	app14sym			hard14sym			hole	urquhart	channel	color
	avg mem	avg time	solved	avg mem	avg time	solved	solved	solved	solved	solved
standard	334MB	597.6s	**113**	323MB	662.9s	106	4	3	2	3
unrelaxed	349MB	611.1s	**113**	336MB	708.8s	107	4	3	2	3
compact	**329MB**	**589.6s**	112	**305MB**	**638.0s**	**108**	4	3	2	3

The theoretical advantage of having a more compact encoding is not translated into a significant increase in the number of solved instances. We do observe average runtime and memory consumption correlating with the size of the encoding, being lowest for the compact encoding and highest for the unrelaxed encoding. We conclude none of the clausal encodings strongly outperforms the others. That said, the compact encoding enjoys a small runtime and memory advantage over both other encodings.

7.2 Row interchangeability and binary clauses

To assess the influence of exploiting row interchangeability and binary clauses, we set up an experiment with four versions of BREAKID:

- BREAKID(): no row interchangeability or binary clauses
- BREAKID(r): version with row interchangeability and without binary clauses
- BREAKID(b): version without row interchangeability and with binary clauses
- BREAKID(r,b): version with both row interchangeability and binary clauses

Each of these versions uses the compact encoding, and limits the lex-leader formulas to 50 auxiliary variables, symmetries used to completely break row interchangeability excepted. The results are summarized in Table 2.

A first observation is that the binary clause improvement shows mixed results, but performs very well on urquhart, allowing all instances to be solved in less than a second. The main reason for the performance degradation is the huge amount of binary clauses derived, amounting over 5 million on some instances. Activating row interchangeability in BREAKID fixes the large number of binary clauses by not allowing variables occurring in row interchangeability matrices to be used in binary clauses.

Table 2. Number of solved instances for BREAKID configurations with and without (r)ow-interchangeability and (b)inary clauses. Also includes average number of corresponding symmetry breaking clauses introduced.

	BREAKID()	BREAKID(b)		BREAKID(r)		BREAKID(r,b)		
	solved	(b) clauses	solved	(r) clauses	solved	(b) clauses	(r) clauses	solved
app14sym	113	37552	111	10245	**114**	190	10245	**114**
hard14sym	108	207719	105	2926	**112**	308	2926	110
hole	4	427	3	1627	**8**	0	1627	**8**
urquhart	3	99	**6**	0	3	99	0	**6**
channel	2	9893	2	15421	**10**	0	15421	**10**
color	3	1469	4	1481	5	656	1481	**6**

The row interchangeability improvement is more successful, improving performance on all benchmark sets except urquhart. Focusing on the pigeonhole, full row interchangeability is detected for all instances, so each instance became polynomially solvable given the presence of symmetry breaking clauses. This is a significant improvement to the preliminary version of BREAKID [8]. A similar effect is seen in the channel routing problem, where activating row interchangeability allows deciding all instances in less than a minute. For the benchmark set as a whole, row interchangeability was detected in 54% of the symmetric instances.

We conclude that row interchangeability exploitation is a significant improvement, while binary clauses have the potential to improve performance on certain types of problems. Furthermore, row interchangeability compensates for weaknesses of the binary clauses approach, and the combination of the two yields the best overall performance.

7.3 Comparison to SHATTER and performance on the 2014 SAT competition

This experiment compares BREAKID to state-of-the-art solving configurations. We use app14, hard14, hole, urquhart, channel and color as benchmark sets. We effectively run all application and hard-combinatorial instances of 2014's SAT competition. The solving configurations used are (with GLUCOSE as SAT engine):

- GLUCOSE: pure GLUCOSE without symmetry breaking.
- SHATTER: SHATTER is run after first deduplicating the input CNF.
- BREAKID: compact encoding, row interchangeability and binary clauses activated.
- BREAKID(100s): same as BREAKID but SAUCY is forced to stop detecting symmetry after 100 seconds of preprocessing have elapsed.

We present the number of instances solved within resource limits, as well as the average time needed to detect symmetry and generate symmetry breaking clauses in Table 3.

First, the two BREAKID variants are the only configurations that handle hole, urquhart and channel efficiently, as SHATTER constructs lex-leader constraints for the wrong set of symmetry generators, and GLUCOSE gets lost in the symmetrical search space for non-trivial instances. A similar conclusion is present for the color instances, though even BREAKID remains unable to solve 4 instances.

On hard14, SHATTER outperforms GLUCOSE, while both BREAKID approaches outperform SHATTER. So for these instances, symmetry detection and breaking is worth

Table 3. Number of solved instances for GLUCOSE, SHATTER, BREAKID and BREAKID limiting SAUCY to 100 seconds. Also includes average preprocessing time in seconds.

	GLUCOSE	SHATTER		BREAKID		BREAKID(100s)	
	solved	pre-time	solved	pre-time	solved	pre-time	solved
hole	2	0.0s	3	0.1s	**8**	0.1s	**8**
urquhart	2	0.1s	2	0.2s	**6**	0.2s	**6**
channel	2	3.2s	2	9.7s	**10**	9.7s	**10**
color	3	2.9s	2	4.1s	**6**	4.1s	**6**
app14	**214**	6.3s	210	74.9s	209	14.7s	211
hard14	164	159.2s	178	181.3s	183	14.8s	**187**

the incurred overhead. The preprocessing time needed by SHATTER is almost completely due to SAUCY's symmetry detection, which exceeds 3600s for 9 instances. BREAKID(100s) solves this problem by limiting the time consumed by SAUCY to 100s, resulting in the best performance on hard14, adding 23 solved instances compared to plain GLUCOSE. Of course, both BREAKID approaches increase the preprocessing overhead by detecting row interchangeability and constructing binary clauses.

As far as app14 is concerned, the benefit of a smaller search space does not outweigh the overhead of detecting symmetry and introducing symmetry breaking clauses.

8 Conclusion

In this paper, we presented novel improvements to state-of-the-art symmetry breaking for SAT. Common themes were to adapt the variable order and the set of generator symmetries by which to construct lex-leader constraints. BREAKID implements these ideas and functions as a symmetry breaking preprocessor in the spirit of SHATTER. Our experiments with BREAKID show the potential for these techniques separately and combined. We observed that BREAKID outperforms SHATTER, and is a particularly effective preprocessor for hard-combinatorial SAT-problems.

The algorithms presented are effective, but also incomplete, e.g., not all row interchangeability is detected, no maximal set of binary symmetry breaking clauses is derived etc. Coupling BREAKID to a computational group algebra system such as GAP [11] has the potential to alleviate these issues.

Alternatively, it might be worth comparing different methods of graph automorphism detection, and investigating how hard it is to adjust their internal search algorithms to put out more useful symmetry generators, stabilizer chains for binary clauses, or even row interchangeability symmetry groups. Jefferson & Petrie already started this research in a constraint programming context [13].

9 Acknowledgement

This research was supported by the project GOA 13/010 Research Fund KU Leuven and projects G.0489.10, G.0357.12 and G.0922.13 of FWO (Research Foundation - Flanders). Bart Bogaerts is supported by the Finnish Center of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (grant #251170).

References

1. Aloul, F., Ramani, A., Markov, I., Sakallah, K.: Solving difficult SAT instances in the presence of symmetry. In: Design Automation Conference, 2002. Proceedings. 39th. pp. 731–736 (2002)
2. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Shatter: efficient symmetry-breaking for boolean satisfiability. In: Design Automation Conference. pp. 836–839 (2003)
3. Aloul, F.A., Sakallah, K.A., Markov, I.L.: Efficient symmetry breaking for Boolean satisfiability. IEEE Transactions on Computers 55(5), 549–558 (2006)
4. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI. pp. 399–404 (2009)
5. Balint, A., Belov, A., Heule, M.J., Järvisalo, M.: The 2013 international SAT competition. satcompetition.org/2013 (2013)
6. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-Breaking Predicates for Search Problems. In: Principles of Knowledge Representation and Reasoning. pp. 148–159. Morgan Kaufmann (1996)
7. Devriendt, J., Bogaerts, B.: BreakID, a symmetry breaking preprocessor for SAT solvers. bitbucket.org/krr/breakid (2015)
8. Devriendt, J., Bogaerts, B., Bruynooghe, M.: BreakIDGlucose: On the importance of row symmetry. In: Proceedings of the Fourth International Workshop on the Cross-Fertilization Between CSP and SAT (CSPSAT) (2014), https://lirias.kuleuven.be/handle/123456789/456639
9. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT. LNCS, vol. 2919, pp. 502–518. Springer (2003)
10. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: Hentenryck, P. (ed.) Principles and Practice of Constraint Programming - CP 2002, LNCS, vol. 2470, pp. 462–477. Springer Berlin Heidelberg (2002), http://dx.doi.org/10.1007/3-540-46135-3_31
11. The GAP Group: GAP – Groups, Algorithms, and Programming, Version 4.7.9 (2015), www.gap-system.org
12. Haken, A.: The intractability of resolution. Theoretical Computer Science 39, 297 – 308 (1985), http://www.sciencedirect.com/science/article/pii/0304397585901446, third Conference on Foundations of Software Technology and Theoretical Computer Science
13. Jefferson, C., Petrie, K.E.: Automatic generation of constraints for partial symmetry breaking. In: Lee, J. (ed.) Principles and Practice of Constraint Programming – CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings. pp. 729–743. Springer Berlin Heidelberg, Berlin, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-23786-7_55
14. Junttila, T., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: Applegate, D., Brodal, G.S., Panario, D., Sedgewick, R. (eds.) Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics. pp. 135–149. SIAM (2007)
15. Katebi, H., Sakallah, K.A., Markov, I.L.: Symmetry and satisfiability: An update. In: Strichman, O., Szeider, S. (eds.) SAT. LNCS, vol. 6175, pp. 113–127. Springer (2010)
16. Lee, J.H.M., Li, J.: Increasing symmetry breaking by preserving target symmetries. In: Milano, M. (ed.) Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings. pp. 422–438. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-33558-7_32

17. Lynce, I., Marques-Silva, J.: Breaking symmetries in sat matrix models. In: Marques-Silva, J., Sakallah, K.A. (eds.) Theory and Applications of Satisfiability Testing – SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings. pp. 22–27. Springer Berlin Heidelberg, Berlin, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-72788-0_6
18. McDonald, I., Smith, B.: Partial symmetry breaking. In: Hentenryck, P. (ed.) Principles and Practice of Constraint Programming - CP 2002: 8th International Conference, CP 2002 Ithaca, NY, USA, September 9–13, 2002 Proceedings. pp. 431–445. Springer Berlin Heidelberg, Berlin, Heidelberg (2002), http://dx.doi.org/10.1007/3-540-46135-3_29
19. McKay, B.D., Piperno, A.: Practical graph isomorphism, {II}. Journal of Symbolic Computation 60(0), 94 – 112 (2014), http://www.sciencedirect.com/science/article/pii/S0747717113001193
20. Puget, J.F.: Breaking symmetries in all-different problems. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI 2005. pp. 272–277 (2005)
21. Sabharwal, A.: Symchaff: exploiting symmetry in a structure-aware satisfiability solver. Constraints 14(4), 478–505 (2009), http://dx.doi.org/10.1007/s10601-008-9060-1
22. Sakallah, K.A.: Symmetry and satisfiability. Handbook of Satisfiability 185, 289–338 (2009)
23. Schaafsma, B., Heule, M.J., Maaren, H.: Dynamic symmetry breaking by simulating zykov contraction. In: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing. pp. 223–236. SAT '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02777-2_22
24. Seress, Á.: Permutation Group Algorithms. Cambridge University Press (2003), http://dx.doi.org/10.1017/CBO9780511546549, cambridge Books Online
25. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. Discrete Appl. Math. 155(12), 1539–1548 (Jun 2007), http://dx.doi.org/10.1016/j.dam.2005.10.018
26. Urquhart, A.: Hard examples for resolution. J. ACM 34(1), 209–219 (Jan 1987), http://doi.acm.org/10.1145/7531.8928
27. Walsh, T.: Symmetry breaking constraints: Recent results. CoRR abs/1204.3348 (2012)